WolfSSL

Luca Valentini

Insert Date

Contents

1 SSL Protocol **1**

1.1 Introduction 1

1.2 SSL Handshake 3

2 Wolf SSL **6**

3 Test case **9**

3.1 Client/Server provided by WolSSL 9

3.2 EchoClient/EchoServer provided by WolfSSL 11

4 Create a program using Wolf SSL **13**

4.1 TCP application 13

4.1.1 TCP Server 13

4.2 From TCP to SSL 18

5 Differences between WolfSSL and OpenSSL **20**

5.1 Introduction 20

Abstract

Explanation of this article. Must be a synthesis

Chapter 1

SSL Protocol

1.1 Introduction

The SSL protool is a client/server protocol that provides the following basic security services to the communicating peers:

- Authentication (both peer entity anda data origin authentication) services
- Connection confidentiality services
- Connection integrity services

The SSL protocol is sockets-oriented, meaning that all or none of the data that is sent to or received from a network connection is cryptographically protected in exactly the same way. It can be best viewed as an intermediate layer between the transporrt and the application layer that serves two purposes:

- Establish a secure connection between the commucating peers
- Use this connection to securely trasmit giher-layer protocol data from the sender to the reciever. It therefore fragments the data in pieces called fragments; each fragment is optionally compressed, authenticated, encrypted, prepended with a header, and transmitted to the reciever. Each data fragment prepared this way is sent in a distinct SSL record.

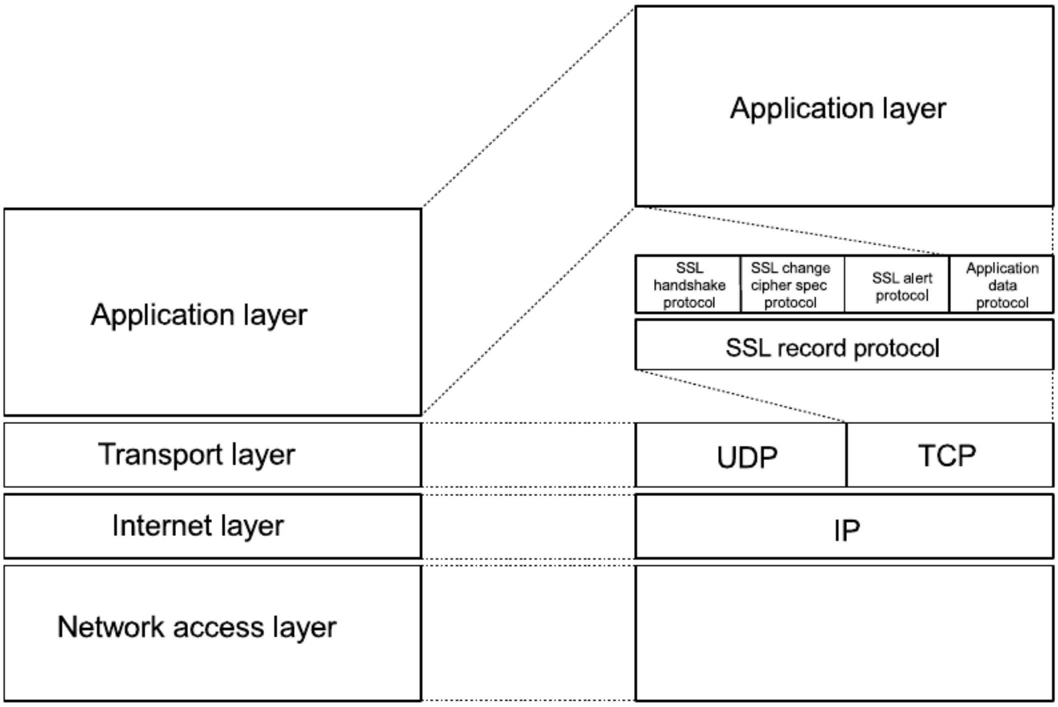


Figure 1.1: The SSL with its (sub)layer and (sub)protocols

The SSL consists of two sublayers and a few subprotocols:

- The lower sublayer is stacked on top of some connection-oriented and reliable transport layer protocol. This layer basically comprises the SSL record protocol that is used for the encapsulation of the higher-layer protocol data.

- The higher sublayer is stacked on top of the SSL record protocol and comprises four subprotocols.

  - The *SSL handshake protocol* is the core subprotocol of SSL. It is used for establishment of a secure connection. It allows the communicating peers to authenticate each other and to negotiate a cipher suite and a compression method.
  - The *SSL change cipher spec protocol* is used to put the parameters, set by the SSL handshake protocol in place and make them effective.
  - The *SSL alert protocol* allows the communicating peers to signal indicators of potential problems and send respective alert messages to each other.

– The *SSL application data protocol* is used for the secure transmission of application data.

In spite of the fact that SSL consists of several subprotocols, we use the term *SSL protocol* to refer to all of them simultaneously.

1.2 SSL Handshake

The SSL handshake protocol is layered on top of the SSL record protocol. It allows a client and server to authenticate each other and to negotiate issues like cipher suites and compression methods.

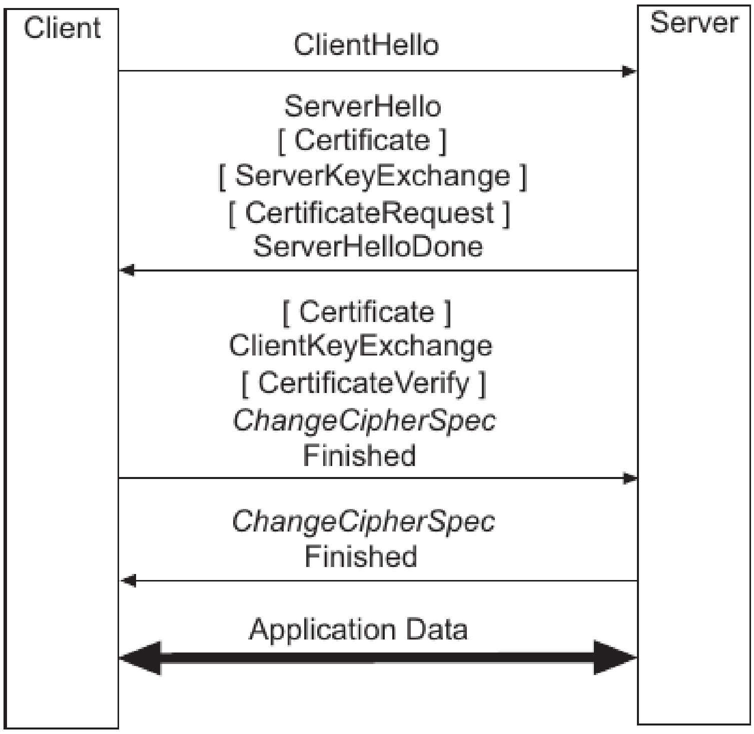


Figure 1.2: The SSL handshake protocol

The SSL handshake protocol comprises four sets of messages:

- The first flight comprises a single *ClientHello* message that is sent from the client to the server.

- The second flight comprises two messages that are sent back from the server to the client:

  1. *ServerHello* message is sent in response to the *ClientHello* message
  2. (optional) If the server is to authenticate itself, it may send a *Certificate* message to the client.
  3. (optional) Under some circumstances, the server may send a *ServerKeyExchange* message to the client.
  4. (optional) If the server requires the client to authenticate itself with a public key certificate, then it may send a *CertificateRequest* message to the client.
  5. Finally, server send a *ServerHelloDone* message to the client.

- The third flight comprises three to five messages that are again sent from the client to the server:

  1. (optional) If the server has sent a *CertificateRequest* message, then the client sends a *Certificate* message to the server.
  2. In the main step of the protocol, the client sends a *ClientKeyExchange* message to the server.
  3. (optional) If the client has sent a certificate to the server, then it must also send a *CertificateVerify* message to the server. This message is digitally signed with the private key that corresponds to the client certificate's public key.
  4. The client sends a *ChangeCipherSpec* message to the server (using the SSL change cipher spec protocol) and copies its pending write state into the current write state.
  5. The client sends a *Finished* message to the server. As mentioned above, this is the first message that is cryptographically protected under the new cipher spec.

- The fourth flight comprises two messages that are sent from the server back to the client:

  1. The server sends another *ChangeCipherSpec* message to the client and copies its pending write state into the current write state.
  2. Finally, the server send a *Finished* message to the client. Again, this message is cryptographically protected under the new cipher spec.

At this point in time, the SSL handshake is complete and the client and server may start exchanging application-layer data.

Chapter 2

Wolf SSL

The wolfSSL embedded SSL library is a lightweight SSL/TLS library written in ANSI C and targeted for embedded, RTOS, and resource-constrained environments - primarily because of its small size, speed, and feature set; It's a SSL/TLS library optimized to run on embedded platforms.
It's free and it has an excellent cross platform support.
WolfSSL supports standards up to the current TLS 1.3 and DTLS 1.2 levels, is up to 20 times smaller than OpenSSL and it's powered by the colfCrypt library.

This library is built for maximum portability and supports the C programming language as a primary interface. It also supports several other host languages, including Java (wolfSSL JNI), C# (wolfSSL C#), Python, and PHP and Perl.

To improve performance it supports hardware cryptography and acceleration on several platforms.

In the following list you can see some of WolfSSI's features:

- Runtime memory usage between 1-36 kB
- OpenSSl compatibility layer
- Hash Functions:
  - MD2
  - MD4
  - MD5
  - SHA-1
  - SHA-224
  - SHA-256
  - SHA-384
  - SHA-512
  - BLAKE2b
  - RIPEMD-160
  - Poly1305

- Mutual authentication support (client/server)
- SSL Sniffer (SSL Inspection) Support
- IPv4 and IPv6 support

The operating systems supported are:

1. Win32/64
2. Linux
3. Mac OS X
4. Solaris
5. ThreadX
6. VxWorks
7. FreeBSD
8. NetBSD
9. OpenBSD
10. embedded Linux
11. Yocto Linux
12. OpenEmbedded
13. WinCE
14. Haiku
15. OpenWRT
16. iPhone(iOS)
17. Android
18. Nintendo Wii and Gamecube through DevKitPro
19. QNX
20. MontaVista
21. NonStop
22. TRON / ITRON / ITRON
23. Micrium C / OS - III
24. FreeRTOS
25. SafeRTOS
26. NXP / Freescale MQX
27. Nucleus
28. TinyOS
29. HP / UX
30. AIX
31. ARC MQX
32. TI - RTOS
33. uTasker
34. embOS
35. INtime
36. Mbed
37. uT - Kernel
38. RIOT
39. CMSIS -RTOS
40. FROSTED
41. Green Hills INTEGRITY
42. Keil RTX
43. TOPPERS
44. PetaLinux
45. Apache Mynewt
46. PikeOS

Chapter 3

Test case

3.1 Client/Server provided by WolSSL

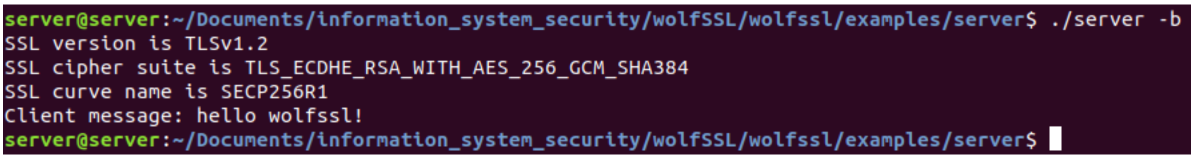


Figure 3.1: Server SSL

+

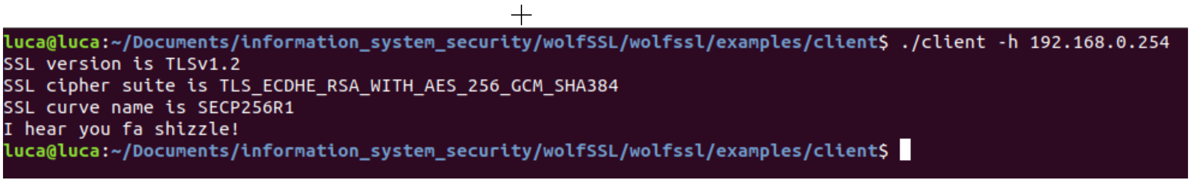


Figure 3.2: Client SSL

In this example, the server is a simple SSL server that allows only one client connection; after the connection with a client, the server receives an encrypted message from client, it responds and quits.
The -b parameter allows the server to bind to any interface instead of localhost only.

The client after the connection with the server, sends a message (hello wolfssl!) and after the server response, it quits.
The -h parameter allows the client to specify the server address to perform the connection.

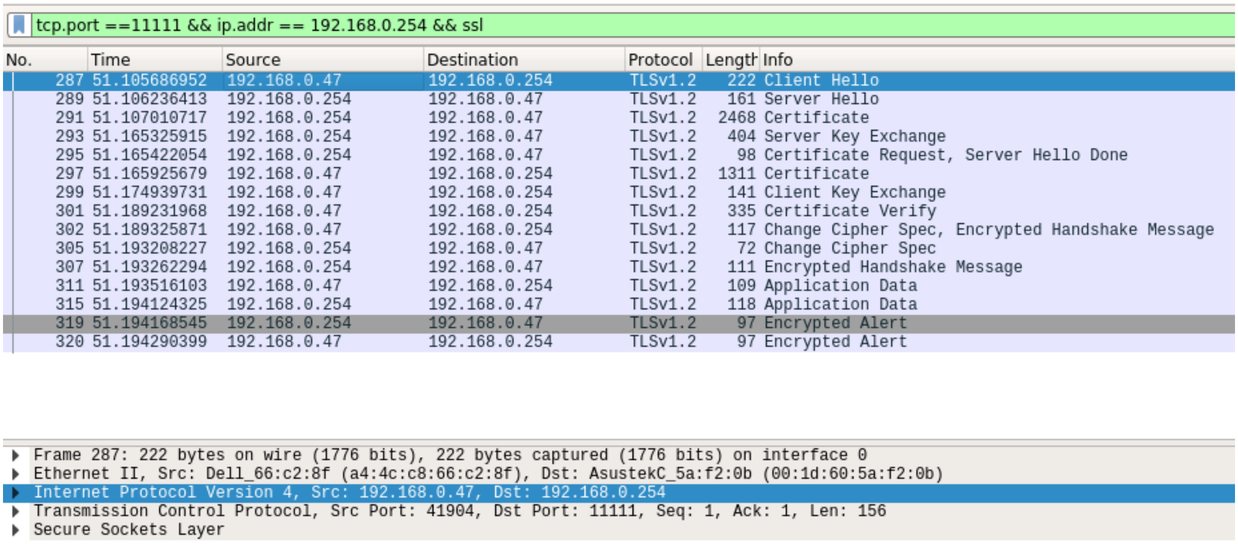


Figure 3.3: All SSL packets sent

Client IP: 192.168.0.47 Server IP: 192.168.0.254

Come si puo' bene vedere dalla figura soprastante, la comunicazione viene iniziata dal client con un 'Client Hello'; dopo SSL handshake, ci sono due messaggi 'Application Data' inviati rispettivamente dal client verso il server e dal server verso il client il cui contenuto e' cifrato. Una volta che il server invia la risposta al client, la comunicazione viene chiusa attraverso 'Encrypted Alert'.

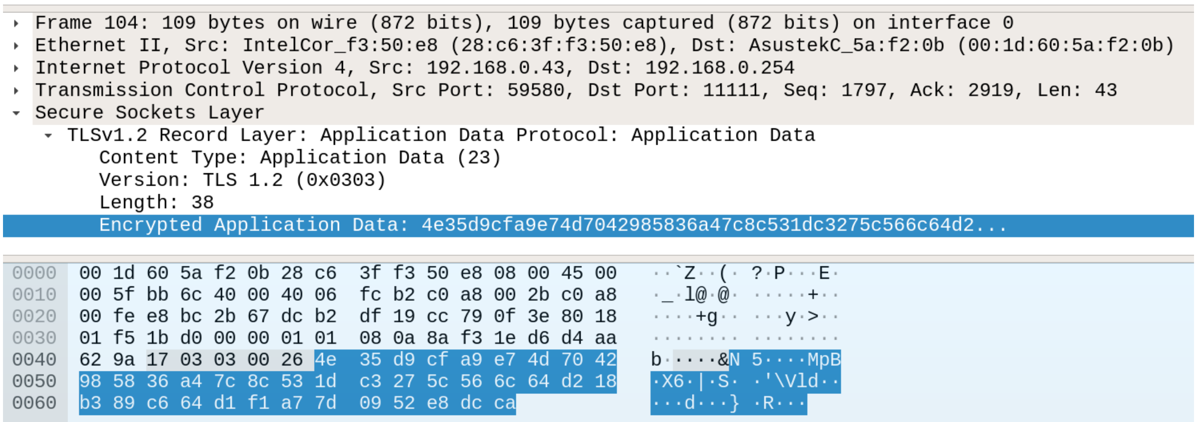
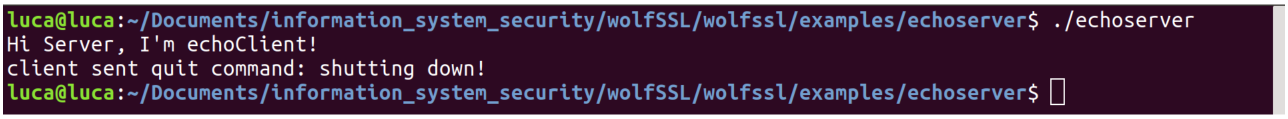


Figure 3.4: Content of the encrypted message

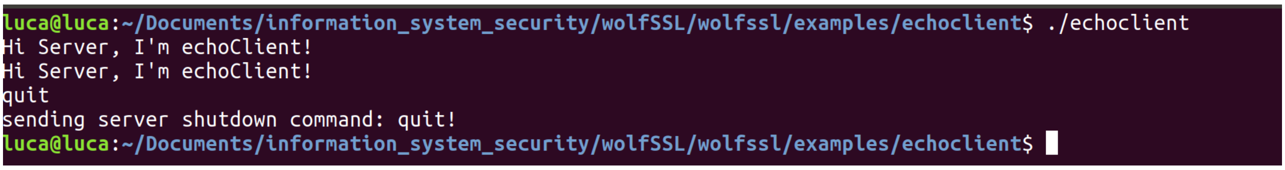
Come si puo' vedere, lo scambio di messaggi e' cifrato.

3.2 EchoClient/EchoServer provided by WolfSSL



```
luca@luca:~/Documents/information_system_security/wolfSSL/wolfssl/examples/echoserver$ ./echoserver
Hi Server, I'm echoClient!
client sent quit command: shutting down!
luca@luca:~/Documents/information_system_security/wolfSSL/wolfssl/examples/echoserver$
```

Figure 3.5: EchoServer SSL



```
luca@luca:~/Documents/information_system_security/wolfSSL/wolfssl/examples/echoclient$ ./echoclient
Hi Server, I'm echoClient!
Hi Server, I'm echoClient!
quit
sending server shutdown command: quit!
luca@luca:~/Documents/information_system_security/wolfSSL/wolfssl/examples/echoclient$
```

Figure 3.6: EchoClient SSL

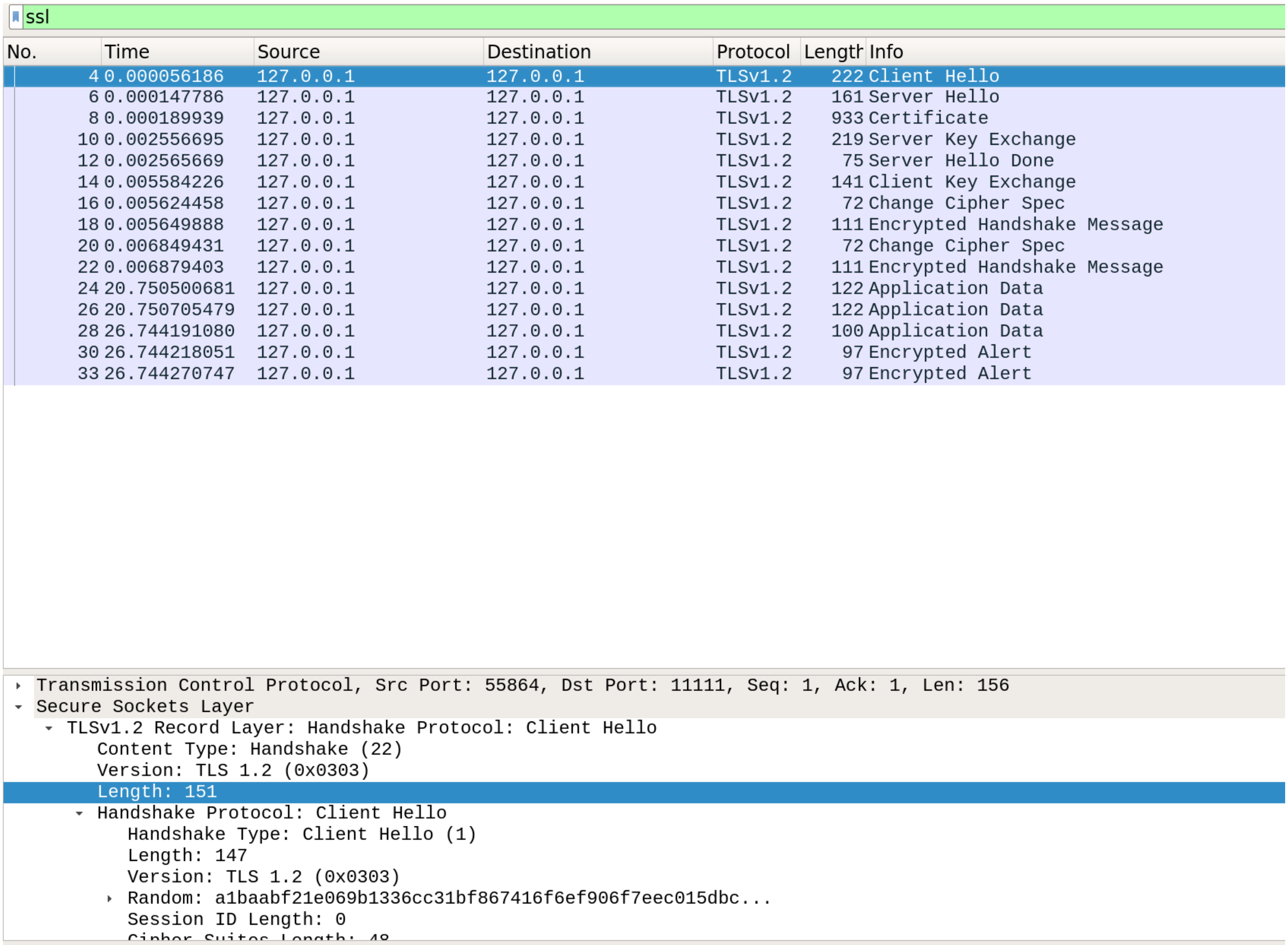


Figure 3.7: All SSL packets sent

Chapter 4

Create a program using Wolf SSL

4.1 TCP application

To create a SSL program you can modify your TCP program added several SSL functions.

4.1.1 TCP Server

In my example, the TCP server after configuring the socket and connecting it with the client, it creates a ClientHandler thread that launches a thread for read and a thread for write from socket. This application creates a chat between a client and a server.

```
int main()
{
    /* Create a socket that uses an internet IPv4 address,
     * Sets the socket to be stream based (TCP),
     * 0 means choose the default protocol. */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        fprintf(stderr, "ERROR: failed to create the socket\n");
        return -1;
    }

    /* Initialize the server address struct with zeros */
    memset(&servAddr, 0, sizeof(servAddr));
```

```
/* Fill in the server address */
servAddr.sin_family = AF_INET;       /* using IPv4    */
servAddr.sin_port = htons(DEFAULT_PORT); /* on DEFAULT_PORT */
servAddr.sin_addr.s_addr = INADDR_ANY; /* from anywhere */

/* Bind the server socket to our port */
if (bind(sockfd, (struct sockaddr *)&servAddr,
    sizeof(servAddr)) == -1)
{
    fprintf(stderr, "ERROR: failed to bind\n");
    return -1;
}

/* Listen for a new connection, allow 5 pending connections */
if (listen(sockfd, 5) == -1)
{
    fprintf(stderr, "ERROR: failed to listen\n");
    return -1;
}

/* Continue to accept clients until shutdown is issued */
while (1)
{
    printf("Waiting for a connection...\n");

    /* Accept client connections */
    if ((connd = accept(sockfd, (struct sockaddr *)&clientAddr,
        &size)) == -1)
    {
        fprintf(stderr, "ERROR: failed to accept the
            connection\n\n");
        ncurses_end();
        return -1;
    }
    pthread_t mainThread;
    pthread_create(&mainThread, NULL, ClientHandler, NULL);
    pthread_join(mainThread, NULL);
    printText("Communication is ended!\n", "System");

    if (is_end)
        break;
}
ncurses_end();
```

```
    printf("Shutdown complete\n");

    /* Cleanup after this connection */
    close(connd);    /* Close the connection to the client */
    /* Cleanup and return */
    close(sockfd);      /* Close the socket listening for clients
       */
    return 0;           /* Return reporting a success          */
}
```

Listing 4.1: int main() of TCP server

As stated above, the clientHandler thread creates two thread, but before it waits the client username.

```
void *ClientHandler(void *args)
{
    int ret;
    /*********************** USERNAME */
    /* Read the client username into our buff array */
    XMEMSET(buff, 0, sizeof(buff));
    ret = read(connd, buff, sizeof(buff));
    ncurses_start();
    clearWin();
    if (ret > 0)
    {
        /* Print to stdout any data the client sends */
        strcpy(username, buff);
        char text[256];
        sprintf(text, "Client %s connected successfully", username);
        printText(text, "System");
        printText("***************************\n", "System");
        fflush(stdout);
    }
    else
    {
        printText("ERROR!!", "System");
        close(sockfd);    /* Close the connection to the server */
        pthread_exit(NULL); /* End theread execution          */
    }
    /**************************** */
    XMEMSET(buff, 0, sizeof(buff));

    if (pthread_create(&Treader, NULL, readBuffer, NULL))
```

```
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
        return NULL;
    }

    if (pthread_create(&Twriter, NULL, writeBuffer, NULL))
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
        return NULL;
    }
    pthread_join(Treader, NULL);
    pthread_join(Twriter, NULL);
    /* Cleanup after this connection */
    close(connd);     /* Close the connection to the client */
    pthread_exit(NULL); /* End theread execution          */
}
```

Listing 4.2: clientHandler thread of TCP server

ReadBuffer is a thread that allows to read messages sent from client. It has an infinite loop that read data from socket open previously; Once it gets the message, with the printText function, the message is printed to the terminal using ncurses.

```
void *readBuffer(void *args)
{
    int ret;
    while (1)
    {
        /* Read the client data into our buff array */
        XMEMSET(buffReader, 0, sizeof(buffReader));
        ret = read(connd, buffReader, sizeof(buffReader));

        if (ret > 0)
        {
            if (!strcmp(buffReader, "quit"))
            {
                pthread_cancel(Twriter);
                pthread_exit(NULL); /* End threaded execution
                         */
            }
            printText(buffReader, username);
```

```
        }
        else
        {
            printText("ERROR READ!!", "System");
            pthread_cancel(Twriter);
            pthread_exit(NULL); /* End threaded execution          */
        }
    }
}
```

Listing 4.3: readBuffer thread of TCP server

WriteBuffer thread has an infinite loop used for write messages from server to client.

```
void *writeBuffer(void *args)
{
    int ret;
    while (1)
    {
        read_in();
        len = XSTRLEN(Rbuffer);

        /* Reply back to the client */
        ret = write(connd, Rbuffer, len);
        printText(Rbuffer, "Server");

        if (XSTRNCMP(Rbuffer, "quit", 4) == 0)
        {
            is_end = 1;
            break;
        }

        if (ret != len)
        {
            printText("ERROR!!", "System");
            break;
        }
    }
    pthread_cancel(Treader);
    return NULL;
}
```

Listing 4.4: writeBuffer thread of TCP server

4.2 From TCP to SSL

To create a wolfSSL application the first thing that I did is include the wolfSSL API header in your program.

```
#include <wolfssl/ssl.h>
```

After the inclusion of the header files, I initialize the library and the WOLFSSL_CTX calling **wolfSSL_Init**; This is necessary to use the library.

The WOLFSSL_CTX structure contains global values for each SSL connection, including certificate information. To create a new WOLFSSL_CTX there is **wolfSSL_CTX_new()** function. it requires an argument which defines the SSL or TLS protocol for the client or server to use. In my case I used TLS 1.3, so the call is:

```
wolfSSL_CTX_new(wolfTLSv1_3_server_method());
```

for the server;

```
wolfSSL_CTX_new(wolfTLSv1_3_client_method())
```

for the client.

In the WOLFSSL_CTX must be loaded the CA (Certificate Authority) so that the client is able to verify the server's identity when they start the connection. To load the CA into the WOLFSSL_CTX there is **wolfSSL_CTX_load_verify_locations()**. This function requires three arguments:

- a WOLFSSL_CTX pointer
- a certificate file
- a path value that point to a directory which should contain CA certificates in PEM format.

this function returns SSL_SUCCESS or SSL_FAILURE.
wolfSSL_CTX_load_verify_locations() can be used for verify the client or the server identity, but in my case, only the client loads the CA, the server uses a self-signed certificate calling:

```
wolfSSL_CTX_use_certificate_file(ctx, CERT_FILE, SSL_FILETYPE_PEM);
```

Also the server private key can be loaded using the wolfSSL library; the function is:

```
wolfSSL_CTX_use_PrivateKey_file(ctx, KEY_FILE, SSL_FILETYPE_PEM)
```

After a TCP connection the WOLFSSL object needs to be created and the file descriptor needs to be associated with the session; the instructions are:

```
//Connect to socket file descriptor
WOLFSSL* ssl;
//create WOLFSSL object
ssl = wolfSSL_new(ctx);
wolfSSL_set_fd(ssl,sockfd);
```

After the previous instructions called by client and server, the server waits an SSL client to initiate the SSL handshake; it waits until a client call **wolfSSL_connect(ssl)** and then the handshake starts.

Chapter 5

Differences between WolfSSL and OpenSSL

5.1 Introduction

The main differences are:

- Memory Usage
  - WolfSSL can be up to 20 times smaller than OpenSSL; The build size is between 20 and 100 KB and the runtime memory usage between 1 and 36 KB. This gives a magior advantage of integrating in smaller embedded devices.
- Hardware Crypto
  - WolfSSL has a partnership with the most MCU manufacturers which allows to be quite early in the market to support hardware acceleration on huge list of platforms.
- Portability
  - WolfSSL is more portable than OpenSSL because is made for real-time, mobile, embedded and enterprise systems.